010091789 **Image available**
WPI Acc No: 1994-359502/199445
XRPX Acc No: N94-281701

Data flow processor - uses charging logic for individual and group-wise programming of mutually orthogonal homogeneously structured cells in integrated circuit chip

Patent Assignee: VORBACH M (VORB-I); PACT INFORMATIONSTECHNOLOGIE GMBH (PACT-N)
Inventor: VORBACH M; MUENCH R

Number of Countries: 001 Number of Patents: 004
Patent Family:

Patent No	Kind	Date	Applicat No	Kind	Date	Week
DE 4416881	A1	19941117	DE 4416881	A	19940513	199445 B
DE 4447706	A1	19970515	DE 4416881	A	19940513	199725
			DE 4447706	A	19940513	
DE 4447707	A1	19970515	DE 4416881	A	19940513	199725
			DE 4447707	A	19940513	
DE 4416881	C2	19980319	DE 4416881	A	19940513	199815

Priority Applications (No Type Date): DE 4316036 A 19930513
Patent Details:

Patent No	Kind	Lan	Pg	Main IPC	Filing Notes
DE 4416881	A1		19	G06F-015/80	
DE 4447706	A1		1	G06F-009/38	Div ex application DE 4416881 Div ex patent DE 4416881
DE 4447707	A1			G06F-009/38	Div ex application DE 4416881 Div ex patent DE 4416881
DE 4416881	C2		22	G06F-015/80	Div in patent DE 4447706 Div in patent DE 4447707

Abstract (Basic): DE 4416881 A

An integrated circuit chip carries a number of mutually-orthogonal homogeneously-structured cells, each with a number of logically and structurally-identical components. The cells are combined in rows and columns, possibly also in groups, for connection to chip input/output connections.

A charging logic is associated with the cells, via which they can be programmed individually and in groups so that optional logical functions and/or networks can be verified. Manipulation of the data flow processor chip's configuration, i.e. the modification of functional parts or macros can be performed during operation without stopping other functional parts or adversely affecting their functioning.

USE/ADVANTAGE - For logical manipulation of data in binary form.

High flexibility and enables parallel operation which is scalable over wide range.

Dwg.1-5/22

Title Terms: DATA; FLOW; PROCESSOR; CHARGE; LOGIC; INDIVIDUAL; GROUP; WISE; PROGRAM; MUTUAL; ORTHOGONAL; HOMOGENEOUS; STRUCTURE; CELL; INTEGRATE; CIRCUIT; CHIP
Derwent Class: T01
International Patent Class (Main): G06F-009/38; G06F-015/80
File Segment: EPI

Manual Codes (EPI/S-X): T01-M05
?

(19) BUNDESREPUBLIK DEUTSCHLAND

DEUTSCHES PATENTAMT

(12) **Patentschrift**

(10) **DE 44 16 881 C 2**

(51) Int. Cl.[6]:
G 06 F 15/80
G 06 F 9/38

(21) Aktenzeichen: P 44 16 881.0-53
(22) Anmeldetag: 13. 5. 94
(43) Offenlegungstag: 17. 11. 94
(45) Veröffentlichungstag der Patenterteilung: 19. 3. 98

DE 44 16 881 C 2

Innerhalb von 3 Monaten nach Veröffentlichung der Erteilung kann Einspruch erhoben werden

(86) Innere Priorität:
P 43 16 036.0 13.05.93

(62) Teil in: P 44 47 706.6
P 44 47 707.4

(73) Patentinhaber:
Pact Informationstechnologie GmbH, 81545 München, DE

(74) Vertreter:
Zahn, R., Dipl.-Ing., Pat.-Anw., 76229 Karlsruhe

(72) Erfinder:
Vorbach, Martin, 76149 Karlsruhe, DE; Münch, Robert, 76149 Karlsruhe, DE

(56) Für die Beurteilung der Patentfähigkeit in Betracht gezogene Druckschriften:

US	48 70 302
EP	05 39 595 A1
WO	90 11 648

(54) Verfahren zum Betrieb einer Datenverarbeitungseinrichtung

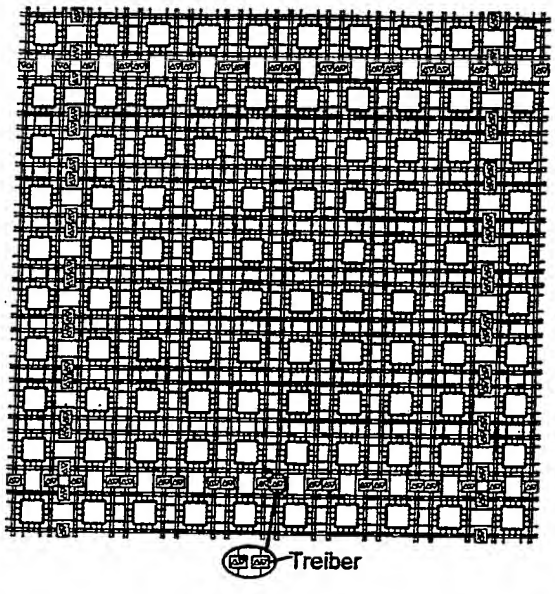
(57) Verfahren zum Betrieb einer Datenverarbeitungseinrichtung mit programmier- und konfigurierbarer Zellstruktur, wobei die Datenverarbeitungseinrichtung eine Zellmatrix aus einer Vielzahl orthogonal zueinander angeordneter, homogen strukturierter Zellen, welche in ihrer Funktion und Vernetzung durch eine Ladelogik frei programmierbar sind, enthält, dadurch gekennzeichnet, daß

a) ein Konfigurationsprogramm, bestehend aus einer Folge von Ladelogik-Befehlen, die jeweils die Funktion und Vernetzung der einzelnen Zellen angeben, so daß sich aus einer Teil folge eine spezielle Konfiguration mit spezieller(en) Anwendung(en) ergibt, vorhanden ist, auf das die Ladelogik (30) Zugriff hat (Fig. 3, Fig. 8, Fig. 14, Fig. 15) und das von ihr ausgeführt wird,

b) zunächst durch eine bestimmte Anzahl von Ladelogik-Befehlen eine Konfiguration (Fig. 20b) mit spezieller(en) Anwendung(en), mittels der Ladelogik (30) zu Beginn als Startkonfiguration in der Zell-Matrix (Fig. 2, Fig. 3) eingestellt wird,

c) durch eine Statemachine (45) oder einen Komparator (48) die Beendigung der Durchführung einer Anwendung einer oder mehrerer Zellen erkannt wird,

d) durch die Erkennung eine Rückmeldung an die Ladelogik (30) erfolgt, die die Programmabarbeitung des Konfigurationsprogramms mit einer anderen Teilfolge (Fig. 21b, Fig. 22b) fortsetzt, welche nur Zellen umkonfiguriert, also neu erstellt, die die Durchführung ihrer Funktion für die aktuelle Anwendung bereits beendet haben oder nicht benötigt werden, so daß die Abarbeitung der Datenströme der an der aktuellen Konfiguration noch beteiligten, parallel (z. B. Register 41, 42) oder gepipelined (z. B. Fig. 22a, Summierer 141 und Multiplizierer 149) arbeitenden Zellen, die sich noch in der Durchführung ihrer Anwendung befinden, nicht gestört wird.



DE 44 16 881 C 2

Beschreibung

Die vorliegende Erfindung bezieht sich auf Verfahren zum Betrieb einer Datenverarbeitungseinrichtung, d. h. einer Hardwareeinheit zur logischen und arithmetischen Manipulation (Verknüpfung) von in binärer Form vorliegenden Daten (Informationen).

Nach dem Stand der Technik sind gewöhnliche Mikroprozessoren (z. B. Intel 80 x 86) bereits bekannt. Diese sind aus fest vorgegebenen Einheiten ausgestaltet und verarbeiten Programme durch das Auskodieren von Befehlen (Microcode) und dadurch bestimmte Registermanipulationen.

Ebenfalls bekannt sind sogenannte FPGAs (z. B. aus US-Patent 4,870,302) die zum Aufbau von komplexen logischen Strukturen verwendet werden. Auf der Grundlage dieser Bausteine lassen sich Rechenwerke wie Addierer, Multiplizierer, etc. innerhalb des Bausteines für die Durchführung einer bestimmten Funktion oder Aufgabe konfigurieren. Durch die direkte Implementierung einer Funktion in die entsprechenden Logikbausteine können FPGAs Funktionen oftmals schneller ausführen als Mikroprozessoren. Obwohl die bekannten Bausteine aufgrund ihrer SRAM-Architektur bereits umkonfiguriert werden können, existiert kein Mechanismus, um die Umkonfiguration schnell und dynamisch während der Laufzeit durchführen zu können, insbesondere, wenn nur Teilbereiche des Bausteines mit einer neuen Funktion konfiguriert werden sollen, während andere Teile des Bausteines ihre Aufgabe fortsetzen. Durch die fehlende Interaktion zwischen einer konfigurierenden Einheit und dem FPGA an sich, scheiden die Bausteine als funktional vollwertiger Ersatz für Mikroprozessoren aus.

Die der vorliegenden Erfindung zugrunde liegende Aufgabe besteht darin, ein Verfahren zum Betrieb einer Datenverarbeitungseinrichtung mit programmierbarer und konfigurierbarer Zell-Struktur — wobei eine Zelle als ein logisches Schaltelement ähnlich US 4,870,302 (L.E.) oder eine besonders ausgestaltete Recheneinheit (ALU) definiert ist — bereitzustellen, das eine höhere Parallelität der Verarbeitung und eine flexiblere Verarbeitung von Daten gewährleistet.

Dabei wird beschrieben, wie Bausteine, die aus einer Vielzahl von zwei- oder mehrdimensionalen Zellstrukturen aufgebaut sind, schnell und effizient nach Ablauf eines Arbeitsschrittes oder Teilarbeitsschrittes, durch Interaktion zwischen dem Baustein und einer konfigurierenden Einheit, dynamisch neu konfiguriert werden, ohne Einfluß auf noch ablaufende Arbeitsschritte zu haben. Die Notwendigkeit einer Umkonfiguration, bzw. das Ende eines Arbeitsschrittes, kann gemäß dieses Verfahrens automatisch erkannt werden. Dadurch kann auf Grundlage dieses Verfahrens ein vollwertiger Ersatz für Mikroprozessoren geschaffen werden.

Ein Vorteil der vorliegenden Erfindung liegt darin, daß die beschriebene Methode eine über einen weiten Raum skalierbare Parallelität ermöglicht. Hierbei wird eine Basis zum schnellen und flexiblen Aufbau von zum Beispiel neuronalen Strukturen geschaffen, wie sie bis dato lediglich mit erheblichem Aufwand durch Software simuliert werden können.

Diese Aufgabe wird durch die im Patentanspruch I angegebenen Merkmale beziehungsweise Verfahrenschritte gelöst. Zur Verdeutlichung der Verfahrensschritte wird beispielsweise ein integrierter Schaltkreis (Chip) mit einer Vielzahl insbesondere orthogonal zueinander angeordneter Zellen mit je einer Mehrzahl jeweils logisch gleicher und strukturell identisch angeordneter Zellen gezeigt, sowie dessen interne Busstruktur, die zur Erleichterung der Programmierung äußerst homogen ist. Grundsätzlich ist es denkbar innerhalb eines Datenflußprozessors Zellen mit verschiedenen Zellogiken und Zellstrukturen unterzubringen, um so die Leistungsfähigkeit zu erhöhen, indem zum Beispiel für Speicheransteuerungen (Businterface) andere Zellen als für arithmetische Operationen (Arithmetisch/logische Einheiten (ALUs) existieren. Insbesondere kann für neuronale Netze eine gewisse Spezialisierung von Vorteil sein. Den Zellen ist eine Ladelogik zugeordnet, über die die Zellen je für sich und gegebenenfalls gruppenweise in sogenannte MACROs (Menge von Zellen, welche zusammen eine definierte Aufgabe lösen) zusammengefaßt so programmierbar sind, daß einerseits beliebige logische Funktionen, andererseits aber auch die Verknüpfung der Zellen untereinander in weiten Bereichen verifizierbar sind. Dies wird erreicht indem jeder einzelnen Zelle ein gewisser Speicherplatz zur Verfügung steht, in dem die Konfigurationsdaten abgelegt sind. Anhand dieser Daten werden Multiplexer oder Transistoren in der Zelle beschaltet um die jeweilige Zellfunktion zu gewährleisten (siehe **Fig. 12**).

Mit anderen als im Patentanspruch 1 gebrauchten Worten besteht der Kern der vorliegenden Erfindung darin, eine Methode für einen Datenflußprozessor vorzuschlagen, der zellular aufgebaut ist und dessen Zellen über eine Ladelogik im arithmetisch-logischen Sinne quasi beliebig neu konfiguriert werden können. Dabei ist es von äußerster Notwendigkeit, daß die betreffenden Zellen einzeln und ohne Beeinflussung der übrigen Zellen oder gar einer Stillegung des gesamten Bausteins umkonfiguriert werden können. Ein Datenflußprozessor gemäß dem vorliegenden Verfahren kann so während eines ersten Arbeitszyklusses als Addierer und während eines späteren Arbeitszyklusses als Multiplizierer "programmiert"/genutzt werden, wobei die Anzahl der für die Addition beziehungsweise die Multiplikation erforderlichen Zellen durchaus unterschiedlich sein können. Dabei bleibt die Plazierung der bereits geladenen und während des Umladeprozesses nicht tangierten MACROs erhalten; der Ladelogik beziehungsweise dem Compiler obliegt es, das neu zu ladende MACRO innerhalb der freien Zellen zu partitionieren (d. h. das zu ladende MACRO so zu zerlegen, daß es sich optimal einfügen läßt). Die Ablaufsteuerung des Programms wird dabei von der Ladelogik übernommen, indem sie gemäß dem momentan ausgeführten Programmabschnitt die entsprechenden MACROs in den Baustein lädt, wobei der Ladevorgang von der später beschriebenen Synchronisationslogik mitgesteuert wird, indem sie den Zeitpunkt des Umladens festlegt. Daher entspricht ein DFP gemäß dem beschriebenem Verfahren nicht der bekannten von-Neumann-Architektur, da die Daten- und Programmspeicher getrennt sind. Dies bedeutet jedoch gleichzeitig eine höhere Sicherheit, da fehlerhafte Programme keinen CODE, sondern lediglich DATEN zerstören können.

Um dem Datenflußprozessor eine arbeitsfähige Struktur zu geben, werden einige Zellen, und zwar unter anderem die Eingabe-/Ausgabefunktionen (I/O) und Speichermanagementfunktionen (I/O) vor dem Laden der Programme geladen und bleiben für gewöhnlich während der gesamten Laufzeit konstant.

Dies ist erforderlich um den Datenflußprozessor an seine Hardwareumgebung anzupassen. Die übrigen Zellen werden zu sogenannten MACROs zusammengefaßt

und können während der Laufzeit nahezu beliebig und ohne Beeinflussung von Nachbarzellen oder anderen MACROs umkonfiguriert werden. Dazu sind die Zellen einzeln und direkt adressierbar.

Um die Umstrukturierung (das Umladen/Umkonfigurieren) der Zellen oder MACROs mit der Ladelogik zu synchronisieren, kann — wo notwendig, da nur Umgeladen werden darf, wenn die MACROs mit ihrer alten Tätigkeit fertig sind — eine Synchronisationsschaltung als MACRO auf dem Datenflußprozessor untergebracht werden, die die entsprechenden Signale an die Ladelogik absendet. Hierzu kann eventuell eine Modifikation der gewöhnlichen MACROs von Nöten sein, da diese dann der Synchronisations-Schaltung Zustandsinformationen zur Verfügung stellen müssen.

Diese Zustandsinformationen signalisieren der Synchronisationslogik für gewöhnlich, daß einzelne MACROs ihre Aufgabe erledigt haben, was aus programmiertechnischer Sicht zum Beispiel die Terminierung einer Prozedur oder das Erreichen der Terminierungsbedingung einer Schleife bedeuten kann. D.h. das Programm wird an einer anderen Stelle fortgesetzt und die die Zustandsinformation absendenden MACROs können umgeladen werden. Zudem kann es von Interesse sein, daß die MACROs in einer bestimmten Reihenfolge umgeladen werden. Hierzu kann eine Wertung der einzelnen Zustandsinformationen durch eine Logik (zum Beispiel einen (Prioritäts-Arbiter) erfolgen. Eine derartige — einfache —. Logik ist in Fig. 19 gezeichnet. Die Logik besitzt sieben Eingangssignale durch die die sieben MACROs ihre Zustandsinformation abgeben. In diesem Fall soll 0 für "in Arbeit" und 1 für "fertig" stehen. Die Logik besitzt drei Ausgangssignale, die an die Ladelogik geführt werden, wobei der Zustand 000 als Ruhezustand gilt. Liegt ein Signal an einem der sieben Eingänge an, so findet eine Dezimal-Binär-Umsetzung statt, so wird zum Beispiel Sync6 als 110 dargestellt, was der Ladelogik anzeigt, daß das MACRO, welches Sync6 bedient, seine Aufgabe beendet hat. Liegen gleichzeitig mehrere Zustandsinformationen am Eingang an, so gibt die Synchronisationsschaltung das Signal mit der höchsten Priorität an die Ladelogik weiter; liegen zum Beispiel Sync0, Sync4 und Sync6 an, so reicht die Synchronistaions-Schaltung zunächst Sync6 an die Ladelogik weiter. Nachdem die entsprechenden MACROs umgeladen sind und somit Sync6 nicht mehr anliegt wird Sync4 weitergeleitet usw. Zur Verdeutlichung dieses Prinzips kann zum Beispiel der Standard-TTL-Baustein 74148 in Betracht gezogen werden.

Über die Ladelogik kann der Datenflußprozessor jeweils optimal und gegebenenfalls dynamisch auf eine zu lösende Aufgabe eingestellt werden. Damit ist zum Beispiel der große Vorteil verbunden, daß neue Normen oder dergleichen einzig und allein durch Umladen des Datenflußprozessors umgesetzt werden können und nicht — wie bisher — einen Austausch mit entsprechendem Anfall von Elektronikschrott bedingen.

Die Datenflußprozessoren sind untereinander kaskadierbar, was zu einer beinahe beliebigen Erhöhung des Parallelisierungsgrades, der Rechenleistung, sowie der Netzgröße in neuronalen Netzen führt. Besonders wichtig ist hier eine klare homogene Verbindung der Zellen mit den Ein-/Ausgangs-Pins (IO-Pins) der Datenflußprozessoren, um möglichst keine Einschränkungen auf die Programme zu haben.

In Fig. 4 ist zum Beispiel die Kaskadierung von vier DFPs gezeigt. Sie erscheinen der Umgebung wie ein großer homogener Baustein (Fig. 5). Prinzipiell sind damit zwei Kaskadierungsmethoden denkbar:

a) Nur die lokalen Verbindungen zwischen den Zellen werden herausgeführt, was im vorliegenden Beispiel zwei IO-Pins pro Kantenzelle und vier IO-Pins pro Eckzelle bedeutet. Allerdings hat der Compiler/Programmierer zu beachten, daß die globalen Verbindungen nicht herausgeführt werden, wodurch die Kaskadierung nicht vollständig homogen ist. (Globale Verbindungen zwischen mehreren Zellen, für gewöhnlich zwischen einer kompletten Zellenreihe oder -spalte — siehe Fig. 1 —; lokale Verbindungen existieren nur zwischen zwei Zellen). Fig. 6a zeigt den möglichen Aufbau innerhalb eines DFPs, Fig. 7a zeigt die daraus resultierende Kaskadierung von mehreren DFPs (drei gezeichnet).
b) Die lokalen und globalen Verbindungen werden herausgeführt, was die Anzahl der benötigten Treiber/IO-Pins und Leitungen drastisch erhöht, in unserem Beispiel (Fig. 6b) auf sechs IO-Pins pro Kantenzelle und zwölf IO-Pins pro Eckzelle. Dadurch ist eine vollständige Homogenität bei der Kaskadierung gegeben (Fig. 7b).

Da die globalen Verbindungen insbesondere bei Verwendung der Kaskadierungstechnik b) sehr lang werden können, kann der unangenehme Effekt auftreten, daß die Zahl der globalen Verbindungen nicht ausreicht, da bekanntlich jede Verbindung nur von einem Signal genutzt werden kann. Um diesen Effekt zu minimieren, kann nach einer gewissen Länge der globalen Verbindungen ein Treiber eingeschleift werden. Der Treiber hat zum einen eine Verstärkung des Signal s zur Aufgabe, die bei langen Strecken und entsprechend hohen Lasten, unbedingt erforderlich ist; zum anderen kann der Treiber in Tristate gehen und damit das Signal unterbrechen. Dadurch können die Abschnitte links und rechts, beziehungsweise oberhalb und unterhalb des Treibers von verschiedenen Signalen genutzt werden, sofern der Treiber in Tristate ist, ansonsten wird ein Signal durchgeschleift. Wichtig ist hierbei, daß die Treiber der einzelnen globalen Leitungen auch einzeln angesteuert werden können, d. h. ein globales Signal kann unterbrochen sein, das Nachbarsignal ist jedoch durchgeschleift. Somit können auf einer globalen Verbindung durchaus abschnittweise verschiedene Signale anliegen, während die globale Nachbarverbindung tatsächlich global von ein und demselben Signal verwendet wird (vergleiche Fig. 18).

Zur besseren Kommunikation zwischen den Datenflußprozessoren und der Ladelogik können sogenannte Shared-Memories eingesetzt werden. So können zum Beispiel Programme von einer Festplatte, die im IO-Bereich eines Datenflußprozessors liegt zur Ladelogik durchgereicht werden, indem die Datenflußprozessoren die Daten von der Platte in den Shared-Memory schreiben und die Ladelogik sie dort abholt. Dies ist besonders wichtig, da hier, wie bereits erwähnt, keine von-Neumann- sondern eine Harvardarchitektur vorliegt. Ebenso sind die Shared-Memories von Vorteil, wenn Konstanten, die im Programm — das im Speicherbereich der Ladelogik liegt — definiert sind, mit Daten — die im Speicherbetrieb der Datenflußprozessoren liegen — verknüpft werden sollen.

Weiterbildungen der vorstehend definierten und umschriebenen Erfindung sind Gegenstand der Unteransprüche.

Eine besondere Verwendung eines Datenflußprozes-

sors gemäß dem beschriebenem Verfahrens ist darin zu sehen, daß er in Verbindung mit geeigneten Ein-/Ausgabe-Einheiten einerseits und einem Speicher andererseits die Basis für einen kompletten (komplexen) Rechner bilden kann. Dabei kann ein Großteil der IO-Funktionen als MACROs auf dem Datenflußprozessor implementiert werden und es brauchen momentan lediglich Spezialbausteine (Ethernet-Treiber, VRAMS...) extern zugefügt zu werden. Bei einer Normänderung oder Verbesserung muß dann wie bereits angedeutet nur das MACRO softwareseitig angepaßt werden; ein Eingriff in die Hardware ist nicht notwendig. Es bietet sich hier an, einen IO-(Eingabe-/Ausgabe-) Stecker festzulegen, über welchen dann die Zusatzbausteine angeschlossen werden können.

Fig. 16 zeigt den stark vereinfachten Aufbau eines heute üblichen Rechners. Durch den Einsatz eines DFP-Bausteins können erhebliche Teile eingespart werden (Fig. 17), wobei die entsprechenden herkömmlichen Baugruppen (CPU, Speicherverwaltung, SCSI-, Tastatur- und Videointerface, sowie der parallelen und seriellen Schnittstellen) als MACROs in die kaskadierten DFPs abgelegt werden. Nur die durch einen DFP nicht nachbildbaren Teile wie Speicher und Leitungstreiber mit nicht TTL-Pegeln oder für hohe Lasten müssen extern zugeschaltet werden. Durch die Verwendung des DFPs ist eine günstige Produktion gegeben, da ein und derselbe Baustein sehr häufig verwendet wird, das Layout der Platine ist durch die homogene Vernetzung entsprechend einfach. Zudem wird der Aufbau des Rechners durch die Ladelogik bestimmt, die hier für gewöhnlich nur zu Beginn der Verarbeitung (nach einem Reset) das DFP-Array lädt, wodurch eine günstige Fehlerkorrektur- und Erweiterungsmöglichkeit gegeben ist. Ein derartiger Rechner kann insbesondere mehrere verschiedene Rechnerstrukturen simulieren, indem einfach der Aufbau des zu simulierenden Rechners in das DFP-Array geladen wird. Zu bemerken ist, daß hierbei der DFP nicht in seiner Funktion als DFP arbeitet sondern lediglich ein hochkomplexes und frei programmierbares Zellarray zur Verfügung stellt, sich hierbei jedoch von herkömmlichen Bausteinen in seiner besonderen guten Kaskadierbarkeit unterscheidet.

Ein weiteres Einsatzgebiet eines solchen Bausteins ist der Aufbau großer neuronaler Netze. Sein besonderer Vorzug liegt hierbei in seiner hohen Gatterdichte, seiner ausgezeichneten Kaskadierbarkeit, sowie seiner Homogenität. Ein Lernvorgang, der eine Änderung einzelner axiomatischer Verbindungen beziehungsweise einzelner Zellfunktionen beinhaltet ist auf üblichen Bausteinen ebenso schlecht durchführbar, wie der Aufbau großer homogener und gleichzeitig flexibler Zellstrukturen. Die dynamische Umkonfigurierbarkeit ermöglicht erstmalig die optimale Simulation von Lernvorgängen.

Die vorliegende Erfindung wird im folgenden anhand der weiteren Figuren näher erläutert. Insgesamt zeigen

Fig. 1 ein aus vier Zellen bestehendes unprogrammiertes SUBMACRO X (analog einem 1-Bit-Addierer gemäß Fig. 12 beziehungsweise Fig. 13) mit den erforderlichen Leitungsanschlüssen;

Fig. 2 einen Teilausschnitt eines integrierten Schaltkreises (Chip) mit einer Vielzahl von Zellen und einem separierten SUBMACRO X gemäß Fig. 1;

Fig. 3 einen integrierten Schaltkreis (Chip) mit einer Orthogonalstruktur einer quasi beliebigen Vielzahl von Zellen und einer extern zugeordneten Ladelogik;

Fig. 4 die Kaskadierung von vier DFPs, wobei die Verbindung zwischen den IO-Pins nur schematisch dargestellt sind (tatsächlich bedeutet eine gezeichnete Verbindung eine Mehrzahl von Leitungen);

Fig. 5 die durch die Kaskadierung erreichte Homogenität;

Fig. 6a die Struktur der E/A-Zellen, wobei die globalen Verbindungen nicht herausgeführt werden,

Fig. 6b die Struktur der E/A-Zellen, jedoch mit herausgeführten globalen Verbindungen;

Fig. 7a die aus Fig. 6a resultierende Kaskadierung, wobei eine Eckzelle, sowie die zwei mit ihr kommunizierenden Treiberzellen der kaskadierten Bausteine (vergleiche hierzu Fig. 4) gezeichnet sind;

Fig. 7b die aus Fig. 6b resultierende Kaskadierung, wobei eine Eckzelle, sowie die zwei mit ihr kommunizierenden Treiberzellen der kaskadierten Bausteine (vergleiche hierzu Fig. 4) gezeichnet sind;

Fig. 8 einen bei spiel haften Aufbau einer Zelle mit Multiplexern zur Auswahl der jeweiligen logischen Bausteine;

Fig. 9 ein Schaltsymbol für einen 8-Bit-Addierer;

Fig. 10 ein Schaltsymbol für einen aus acht 1-Bit-Addierern bestehenden 8-Bit-Addierer nach Fig. 9;

Fig. 11 eine logische Struktur eines 1-Bit-Addierers entsprechend Fig. 10;

Fig. 12 eine Zellenstruktur des 1-Bit-Addierers entsprechend Fig. 11;

Fig. 13 einen der Zellenstruktur nach Fig. 9 entsprechend aufgebauten 8-Bit-Addierer;

Fig. 14 ein erstes Ausführungsbeispiel einer Mehrzahl miteinander zu einem Rechenwerk gekoppelter integrierter Schaltkreise (Datenflußprozessor) nach Fig. 3;

Fig. 15 ein zweites Ausführungsbeispiel einer Mehrzahl miteinander zu einem Rechenwerk gekoppelter integrierter Schaltkreise (Datenflußprozessor) nach Fig. 3;

Fig. 16 den stark schematisierten Aufbau eines herkömmlichen Rechners;

Fig. 17 den möglichen Aufbau desselben Rechners mit Hilfe eines Arrays aus kaskadierten DFPs;

Fig. 18 einen Ausschnitt mit eingezeichneten (Leitungs-) Treibern eines DFPs.

Fig. 19 eine zum Beispiel mit einem Standard-TTL-Baustein 74148 ausgeführte Synchronisationslogik;

Fig. 20a, b, c ein Ausführungsbeispiel eines MACRO zur Addition zweier Zahlenreihen;

Fig. 21a eine Multiplikationsschaltung (vergleiche Fig. 20);

Fig. 21b die interne Struktur des DFPs nach dem Laden (vergleiche Fig. 20b);

Fig. 21c die Arbeitsweise des DFPs im Speicher, sowie die Zustände der Zähler 47, 49;

Fig. 22a, b, c eine Kaskadenschaltung, wobei der Addierer aus Fig. 20 und der Multiplizierer aus Fig. 21 zur Steigerung der Rechenleistung hintereinander geschaltet sind;

In Fig. 9 ist ein Schaltsymbol eines 8-Bit-Addierers dargestellt. Das Schaltsymbol besteht aus einem quadratischen Baustein 1 mit acht Eingängen A 0...7 für ein erstes Datenwort A und acht Eingängen B 0...7 für ein zweites (zu addierendes) Datenwort B. Die jeweils acht Eingänge A_i, B_i ($i = 0...7$) werden ergänzt durch einen weiteren Eingang Üein über den dem Baustein 1 gegebenenfalls ein Übertrag zugeleitet wird. Der Baustein 1 hat funktions- und bestimmungsgemäß acht Ausgänge S 0...7 für binären Summanden und einen weiteren Ausgang Üaus für den gegebenenfalls bestehenden Übertrag.

Das in Fig. 9 dargestellte Schaltsymbol ist in Fig. 10

als Anordnung sogenannter SUBMACROS dargestellt. Diese SUBMACROS 2 bestehen je aus einem 1-Bit-Addierer 3 mit je einem Eingang für die entsprechenden Bits des Datenworts und einem weiteren Eingang für ein Übertragsbit. Die 1-Bit-Addierer 3 weisen darüberhinaus einen Ausgang für den Summanden und einen Ausgang für den Übertrag Üaus auf.

In Fig. 11 ist die binäre Logik eines 1-Bit-Addierers beziehungsweise eines SUBMACROS 3 nach Fig. 10 dargestellt. Analog zu Fig. 10 weist diese Schaltlogik je einen Eingang A_i, B_i für die konjugierten Bits der zu verknüpfenden Daten auf; ferner ist ein Eingang Üein für den Übertrag vorgesehen. Diese Bits werden den dargestellten Verbindungen beziehungsweise Verknüpfungen entsprechend in zwei ODER-Gliedern 5 und drei NAND-Gliedern 6 verknüpft, so daß am Ausgangsanschluß Si und am Ausgang für den Übertrag Üaus die einem Volladdierer entsprechenden Verknüpfungsergebnisse (Si, Üaus) anstehen.

Auf der Grundlage logisch und strukturell identischer Zellen 10, deren einzelne logische Bausteine der auszuführenden Verknüpfungsfunktion entsprechend verschaltet werden, wird der 8-bit Addierer 2 in geeigneter Weise in die Zellstruktur implementiert. Der Vorgang geschieht mittels der noch zu beschreibenden Ladelogik. Gemäß der in Fig. 12 gezeigten, von der Schaltlogik nach Fig. 11 abgeleiteten Verknüpfungslogik für einen 1-Bit-Addierer sind je zwei Zellen 10.1, 10.2 bezüglich der logischen Bausteine insoweit gleich, daß jeweils ein ODER-Glied 5 und ein NAND-Glied 6 aktiviert sind. Eine dritte Zelle 10.3 wird nur als Leitungszelle (Leiterbahnzelle) benutzt und die vierte Zelle 10.4 ist bezüglich des dritten NAND-Gliedes 6 aktiv geschaltet. Das aus den vier Zellen 10.1.... 10.4 bestehende SUBMACRO 2 steht somit stellvertretend für einen 1-Bit-Addierer, d. h. ein 1-Bit-Addierer einer Datenverarbeitungseinrichtung nach Art der vorliegenden Erfindung kann über vier entsprechend programmierte (konfigurierte) Zellen 10.1. ... 10.4 verifiziert werden. (Der Vollständigkeit halber soll angemerkt werden, daß die einzelnen Zellen ein erheblich umfangreicheres Netzwerk von logischen Bausteinen, sprich Verknüpfungsgliedern, und Invertern aufweist, die jeweils dem aktuellen Befehl der Ladelogik zufolge aktiv geschaltet werden können. Neben den logischen Bausteinen ist auch ein dichtes Netz von Verbindungsleitungen zwischen den jeweils benachbarten Bausteinen und zum Aufbau von zeilen- und spaltenweisen Busstrukturen zur Datenübertragung andererseits vorgesehen, so daß über eine entsprechende Programmierung seitens der Ladelogik quasi beliebige logische Verknüpfungsstrukturen implementiert werden können).

Der Vollständigkeit halber ist in Fig. 13 der Zellenaufbau eines 8-Bit-Addierers in seiner Gesamtheit dargestellt. Die in Fig. 13 gezeigte Struktur entspricht insoweit der nach Fig. 10, wobei die in Fig. 10 symbolisch als SUBMACROS 3 dargestellten 1-Bit-Addierer jeweils durch eine vier-zellige Einheit 10.1.... 10.4 ersetzt sind. Bezogen auf einen Datenflußprozessor gemäß des beschriebenen Verfahrens bedeutet dies, daß zweiunddreißig Zellen der zur Verfügung stehenden Gesamtheit von Zellen einer zellular mit logisch identischem Layout gefertigten Schaltungsplatine seitens der Ladelogik so angesteuert und konfiguriert beziehungsweise programmiert werden, daß diese zweiunddreißig Zellen ein 8-Bit-Addierer bilden.

In der Darstellung nach Fig. 13 ist über eine strichpunktierte Umrahmung ein SUBMACRO "X" zeichnerisch separiert, das letztlich als aus vier einem 1-Bit-Addierer entsprechend programmierten Zellen (10 gemäß Fig. 12) bestehende Untereinheit zu betrachten ist.

Das in Fig. 13 separierte SUBMACRO "X" ist in Fig. 1 als Teil eines integrierten Schaltkreises (Chip) 20 gemeinsam mit Leitungs- und Datenanschlüssen dargestellt. Das SUBMACRO "X" besteht aus den vier Zellen 10 die entsprechend der orthogonalen Struktur je Seite vier Datenanschlüsse (also insgesamt sechzehn Datenanschlüsse je Zelle) aufweisen. Die Datenanschlüsse verbinden jeweils benachbarte Zellen, so daß ersichtlich wird, wie beispielsweise eine Dateneinheit von Zelle zu Zelle durchgeschleust wird. Die Ansteuerung der Zellen 10 erfolgt einerseits über sogenannte lokale Steuerungen, das sind lokale Leitungen, die mit allen Zellen verbunden sind, und andererseits über sogenannte globale Leitungen, d. h. Leitungen, die über den gesamten integrierten Schaltkreis (Chip) 20 geführt sind.

In Fig. 2 ist ein vergrößerter Ausschnitt eines integrierten Schaltkreises 20 dargestellt, der mit einem orthogonalen Raster von Zellen 10 belegt ist. Wie in Fig. 2 angedeutet kann so zum Beispiel eine Gruppe von vier Zellen 10 als SUBMACRO "X" ausgewählt und dem 1-Bit-Addierer entsprechend Fig. 12 gemäß programmiert beziehungsweise konfiguriert werden.

Ein vollständiger integrierter Schaltkreis gemäß dem beschriebenen Verfahren (DFP) 20 ist beispielsweise in Fig. 3 dargestellt. Dieser integrierte Schaltkreis 20 besteht aus einer Vielzahl im orthogonalen Raster angeordneter Zellen 10 und weist an seinen Außenkanten eine entsprechende Anzahl von Leitungsanschlüssen (Pins) auf, über die Signale, insbesondere Ansteuersignale und Daten zugeführt und weitergeleitet werden können. In Fig. 3 ist wiederum das SUBMACRO "X" gemäß Fig. 13/Fig. 1 abgegrenzt; darüberhinaus sind auch weitere SUBMACROS separiert, die spezifischen Funktionen und Vernetzungen entsprechend zu Untereinheiten zusammengefaßt sind. Dem integrierten Schaltkreis (Chip) 20 ist eine Ladelogik 30 zugeordnet beziehungsweise übergeordnet, über die der integrierte Schaltkreis 20 programmiert und konfiguriert wird. Die Ladelogik 30 teilt letztlich dem integrierten Schaltkreis 20 mit, wie er arithmetisch-logisch zu arbeiten hat.

Anhand von Fig. 14 beziehungsweise Fig. 15 soll im folgenden eine Rechnerstruktur beschrieben werden, die auf den im vorstehenden definierten und erläuterten integrierten Schaltkreis 20 aufbaut.

Gemäß dem in Fig. 14 dargestellten ersten Ausführungsbeispiel ist — analog zur Anordnung der Zellen — im Orthogonalraster eine Mehrzahl von integrierten Schaltkreisen 20 angeordnet, deren jeweils benachbarte über lokale BUS-Leitungen 21 miteinander gekoppelt beziehungsweise vernetzt sind. Die — beispielsweise aus sechzehn integrierten Schaltkreisen 20 bestehende — Rechnerstruktur weist Ein-/Ausgangsleitungen IO auf, über die der Rechner quasi mit der Außenwelt in Verbindung steht, d. h. korrespondiert. Der Rechner gemäß Fig. 14 weist ferner einen Speicher 22 auf, der dem dargestellten Ausführungsbeispiel entsprechend aus zwei separierten Speichern, zusammengesetzt aus jeweils RAM, ROM sowie einem Dual-Ported RAM als shared memory zu der Ladelogik geschaltet, besteht, die gleichermaßen als Schreib-Lese-Speicher oder auch nur als Lese-Speicher realisiert sein können. Der soweit beschriebenen Rechnerstruktur ist die Ladelogik 30 zubeziehungsweise übergeordnet, mittels der die integrierten Schaltkreise (Datenflußprozessor) 20 programmiert und konfiguriert und vernetzt werden.

Die Ladelogik 30 baut beispielsweise auf einem Transputer 31, d. h. einem Prozessor mit mikrocodiertem Befehlssatz auf, dem seinerseits ein Speicher 32 zugeordnet ist. Die Verbindung zwischen dem Transputer 31 und dem Datenflußprozessor basiert auf einer Schnittstelle 33 für die sogenannten Ladedaten, d. h. die Daten die den Datenflußprozessor aufgabenspezifisch programmieren und konfigurieren und einer Schnittstelle 34 für den bereits genannten Rechnerspeicher 22, d. h. den Shared-Memory-Speicher.

Die in Fig. 14 dargestellte Struktur stellt so einen kompletten Rechner dar, der über die Ladelogik 30 jeweils fall- beziehungsweise aufgabenspezifisch programmiert und konfiguriert werden kann. Der Vollständigkeit halber sei noch angemerkt, daß — wie in Verbindung mit der Ladelogik 30 über Pfeile angedeutet — mehrere dieser Rechner vernetzt, d. h. miteinander gekoppelt werden können.

Ein weiteres Ausführungsbeispiel einer Rechnerstruktur ist in Fig. 15 dargestellt. Im Unterschied zu Fig. 14 sind dabei neben den lokalen BUS-Leitungen zwischen den benachbarten integrierten Schaltkreisen 20 noch übergeordnete zentrale BUS-Leitungen 23 vorgesehen, um zum Beispiel spezifische Ein- beziehungsweise Ausgangsprobleme lösen zu können. Auch der Speicher 22 (Shared-Memory) ist über zentrale BUS-Leitungen 23 mit den integrierten Schaltkreisen 20 verbunden, und zwar wie dargestellt jeweils mit Gruppen dieser integrierten Schaltkreise. Die in Fig. 15 dargestellte Rechnerstruktur weist die gleiche Ladelogik 30 auf, wie sie anhand von Fig. 14 erläutert wurde.

In Verbindung mit Fig. 20a soll eine aus erfindungsgemäßen Datenflußprozessoren aufgebaute Additionsschaltung erläutert werden. Ausgegangen wird von zwei Zahlenreihen A_n und B_n für sämtliche n zwischen 0 und 9; die Aufgabe besteht darin, die Summe $C_i = A_i + B_i$ zu bilden, wobei der Index i die Werte $0 \leq n < 9$ annehmen kann.

Bezugnehmend auf die Darstellung nach Fig. 20a ist die Zahlenreihe A_n in einem ersten Speicher RAM1 abgespeichert und zwar zum Beispiel ab einer Speicheradresse 1000 h; die Zahlenreihe B_n ist in einem Speicher RAM2 an einer Speicheradresse 0dfa0h abgespeichert; die Summe C_n wird in den RAM1 eingeschrieben und zwar unter der Adresse 100ah.

Es ist ein weiterer Zähler 49 zugeschaltet, der lediglich die einzelnen durch die Steuerschaltung freigegebenen Taktzyklen hochzählt. Dies soll im Weiteren zur Verdeutlichung der Umkonfigurierbarkeit einzelner MACROs ohne Beeinflussung der an der Umkonfigurierung nicht beteiligten MACROs dienen.

Fig. 20a zeigt zunächst die eigentliche Additionsschaltung 40, die aus einem ersten Register 41 zur Aufnahme der Zahlenreihe A_n und einem zweiten Register 42 zur Aufnahme der Zahlenreihe B_n besteht. Den beiden Registern 41/42 ist ein 8-Bit-Addierer entsprechend dem in Fig. 9 dargestellten MACRO 1 nachgeschaltet. Der Ausgang des MACRO 1 führt über eine Treiberschaltung 43 zurück zum Speicher RAM1. Die Taktbeziehungsweise Zeitsteuerung der Additionsschaltung 40 erfolgt über eine von einem Taktgenerator T angesteuerte Zustandsmaschine (STATEMACHINE) 45, die mit den Registern 41, 42 und der Treiberschaltung 43 verbunden ist.

Die Additionsschaltung 40 wird funktional durch eine Adreßschaltung 46 zur Generierung der Adreßdaten für die abzuspeichernden Additionsergebnisse ergänzt. Die Adreßschaltung 46 besteht ihrerseits aus drei MACROs 1 (gemäß Fig. 9) zur Bildung der Adreßdaten, wobei diese MACROs 1 wie folgt geschaltet sind: Über jeweils einen Eingang werden die zu verknüpfenden Adressen für A_n, B_n, C_n zugeführt. Diese Adressen werden mit den Ausgangssignalen eines Zählers 47 addiert und mit der Statemachine 45 so verknüpft, daß am Ausgang die neue Zieladresse ansteht. Der Zähler 47 und der Komparator 48 haben dabei die Aufgabe sicherzustellen, daß jeweils die richtigen Summanden verknüpft werden und daß jeweils am Ende der Zahlenreihen, d. h. bei n = 9 abgebrochen wird. Ist die Addition vollendet, so wird in der Zustandsmaschine 45 ein STOP-Signal generiert und die Schaltung passiv geschaltet. Ebenso kann das STOP-Signal als Eingangssignal für eine Synchronisations-Schaltung verwenden werden, indem die Synchronisationslogik anhand dieses Signals erkennen kann, daß die Gesamtfunktion "Addieren" gemäß dem nachfolgend beschriebenen ML1 Programm beendet ist und die MACROs somit durch Neue ersetzt werden können (zum Beispiel könnte STOP das Signal Sync5 sein).

Der Zeitablauf in der 45 (STATEMACHINE) läßt sich dabei wie folgt darstellen, wobei noch anzumerken ist, daß in der Zustandsmaschine 45 eine Verzögerungszeit T (in Form von Taktzyklen) zwischen der Adreßgenerierung und dem Datenerhalt implementiert ist:

- Im Zyklus 1 wird jeweils der Zähler 47 um 1 erhöht und im Komparator 48 wird geprüft, ob $n > 9$ erreicht ist; synchron zu diesen Operationen werden die Adressen für A, B, C berechnet;
- im Zyklus (T + 1) werden die Summanden A, B ausgelesen und addiert;
- im Zyklus (T + 2) wird die Summe C abgespeichert.

Mit anderen Worten heißt dies, daß die Operationsschleife und die eigentliche Addition gerade (T + 2) Taktzyklen erfordert. Im allgemeinen sind für T 2..3 Takte erforderlich, so daß verglichen mit den herkömmlichen Prozessoren (CPU), die im allgemeinen 50 bis mehrere 100 Taktzyklen bedingen, eine ganz wesentliche Rechenzeit-Reduzierung möglich wird.

Die anhand von Fig. 20 aufgezeigte Konfiguration soll im folgenden über eine hypothetische MACRO-Sprache ML1 nochmals erläutert werden:

Es existieren die Zahlenreihen A_n und B_n
n: $0 \leq n < = 9$
Es sollen die Summen $C_i = A_i + B_i$ mit $i \in N$ gebildet werden.
const n = 9;
array A[n] in RAM[1] at 1000 h;
array B[n] in RAM[2] at 0dfa0h;
array C[n] in RAM[1] at 100ah;
for i = 0 to n with (A[i], B[i], C[i])
Δ1;
C = Δ1 = A + B;
next;
RAM[1] ist der 1. Speicherblock
RAM[2] ist der 2. Speicherblock
at folgt die Basisadresse der Arrays
for ist der Schleifenbeginn
next ist das Schleifenende
with ( ) folgen die Variablen, deren Adressen durch die Zählvariable i bestimmt werden
T folgt die Verzögerungszeit für eine Statemachine in Taktzyklen
Das Timing der Zustandsmaschine (Statemachine) sieht demnach folgendermaßen aus:

Zyklus Aktivität
1 Zähler erhöhen, Vergleich auf > 9 (ja ⇒ Abbruch) und
Adressen für A, B, C, berechnen
T + 1 A, B, holen und addieren
T + 2 Nach C speichern
Das heißt — wie bereits erwähnt — die Schleife und die Addition benötigen gerade einmal T + 2 Taktzyklen.

Fig. 20b zeigt den groben Aufbau der einzelnen Funktionen (MACROs) in einem erfindungsgemäßen DFP. Die MACROs sind in ihrer etwaigen Lage und Größe eingezeichnet und mit den anhand von Fig. 20a erläuterten entsprechenden Nummern versehen.

Fig. 20c zeigt den groben Aufbau der einzelnen Funktionen auf die RAM-Blöcke 1 und 2: Die Summanden werden nacheinander in aufsteigender Reihenfolge aus den RAM-Blöcken 1 und 2 ab Adresse 1000 h beziehungsweise 0dfa0h gelesen und in RAM-Block 1 ab Adresse 100ah gespeichert. Zudem sind die Zähler 47 und 49 gegeben, beide zählen während des Ablaufs der Schaltung von 0 bis 9.

Nach Beendigung des beschriebenen Programms soll ein neues Programm geladen werden, das die Ergebnisse weiterverwertet. Die Umladung soll zur Laufzeit erfolgen. Das Programm ist im Folgenden gegeben:

Es existieren die Zahlenreihen A_n und B_n, wobei A_n durch das Ergebnis C_n des vorher ausgeführten Programms gegeben ist:

$n: 0 \leq n < = 9$

Es sollen die Produkte $C_i = A_i \times B_i$ mit $i \in N$ gebildet werden.

```
const n = 9
array A[n] in RAM[1] at 100ah
array B[n] in RAM[2] at 0dfa0h
array C[n] in RAM[1] at 1015 h
for i = 0 to n with (A[i], B[i], C[i])
Δ1;
C = Δ1 = A × B;
next.
```

Die Beschreibung der einzelnen Befehle ist bereits bekannt, × symbolisiert die Multiplikation.

Die MACRO-Struktur ist in Fig. 21a beschrieben, Fig. 21b gibt in bekannter Weise die Lage und Größe der einzelnen MACROs auf dem Chip an, besonders zu beachten ist die Größe des Multiplizierers 2 in Vergleich zu Addierer 1 aus Fig. 20b. In Fig. 21c ist erneut die Auswirkung der Funktion auf den Speicher aufgezeigt, Zähler 47 zählt erneut von 0 bis 9, d. h. er wird beim Nachladen der MACROs zurückgesetzt.

Besonders zu beachten ist der Zähler 49. Angenommen, das Umladen der MACROs beträgt 10 Taktzyklen. Dann läuft der Zähler 49 von 9 auf 19, da der Baustein dynamisch umgeladen wird, d. h. nur die umzuladenden Teile werden gestoppt, der Rest arbeitet weiter. Das führt nun dazu, daß der Zähler während des Programmablaufs von 19 auf 29 hochläuft. (Hiermit soll das dynamische unabhängige Umladen demonstriert werden, in jedem bisher bekannten Baustein würde der Zähler erneut von 0 auf 9 laufen, da er zurückgesetzt wird).

Bei näherer Betrachtung des Problems stellt sich die Frage, warum nicht beide Operationen, die Addition und die Multiplikation in einem Zyklus durchgeführt werden, also die Operation:

Es existieren die Zahlenreihen A_n und B_n, wobei A_n durch das Ergebnis von C_n des vorher ausgeführten Programms gegeben ist:

$n: 0 \leq n < = 9$

Es sollen die Produkte $C_i = (A_i + B_i) \times B_i$ mit $i \in N$ gebildet werden.

```
path D;
const n = 9;
array A[n] in RAM[1] at 1000 h
array B[n] in RAM[2] at 0dfa0h
array C[n] in RAM[1] at 100ah
for i = 0 to n with (A[i], B[i], C[i])
Δ1;
D = Δ1 = A + B;
C = Δ1 = D × B;
next;
```

path D definiert einen internen nicht aus den DFP herausgeführten Doppelpfad. Die Operation benötigt wegen einem zusätzlichen Δ1 einen Taktzyklus mehr als vorher, ist insgesamt jedoch schneller als die beiden obigen Programme in Folge ausgeführt, da zum einen die Schleife nur einmal durchlaufen wird, zum zweiten nicht umgeladen wird.

Prinzipiell könnte das Programm auch so formuliert werden:

```
const n = 9;
array A[n] in RAM[1] at 1000 h
array B[n] in RAM[2] at 0dfa0h
array C[n] in RAM[1] at 100ah
for i = 0 to n with (A[i], B[i], C[i])
Δ1;
C = Δ2 = (A + B) × B;
next;
```

Sind die Gatterlaufzeiten des Addierers und des Multiplizierers zusammen kleiner als ein Taktzyklus, kann die Operation $(A + B) \times B$ auch in einem Taktzyklus durchgeführt werden, was zu einer weiteren erheblichen Geschwindigkeitssteigerung führt:

```
const n = 9;
array A[n] in RAM[1] at 1000 h
array B[n] in RAM[2] at 0dfa0h
array C[n] in RAM[1] at 100ah
for i = 0 to n with (A[i], B[i], C[i])
Δ1;
C = Δ1 = (A + B) × B;
next;
```

Anhand von Fig. 8 soll ein einfaches Beispiel eines Zellenaufbaus erläutert werden. Die Zelle 10 umfaßt zum Beispiel ein UND-Glied 51, ein ODER-Glied 52, ein XOR-Glied 53, einen Inverter 54 sowie eine Registerzelle 55. Die Zelle 10 weist darüberhinaus eingangsseitig zwei Multiplexer 56, 57 mit (den sechzehn Eingängen der Zelle entsprechend Fig. 1) zum Beispiel je sechzehn Eingangsanschlüssen IN1, IN2 auf. Über diesen (16 : 1)-Multiplexer 56/57 werden jeweils die den genannten logischen Gliedern UND, ODER, XOR 51...53 zuzuführenden Daten ausgewählt. Diese logischen Glieder sind ausgangsseitig mit einem (3 : 1)-Multiplexer 58 gekoppelt, der seinerseits mit dem Eingang des Inverters 54, einem Eingang der Registerzelle 55 und einem weiteren (3 : 16)-Multiplexer 59 gekoppelt ist. Der letztgenannte Multiplexer 59 ist zusätzlich mit dem Ausgang des Inverters 54 und einem Ausgang der Registerzelle 55 verbunden und gibt das Ausgangssignal OUT ab.

Der Vollständigkeit halber sei angemerkt, daß die Registerzelle 55 mit einem Reset-Eingang R und einem Takteingang gekoppelt ist.

Dem im vorstehenden erläuterten Zellenaufbau, d. h. der Zelle 10 ist nun eine Ladelogik 30 übergeordnet, die mit den Multiplexern 56, 57, 58 und 59 verbunden ist und diese den gewünschten Funktionen entsprechend ansteuert.

Sollen zum Beispiel die Signale A2 mit B5 verrundet

werden, so werden die Multiplexer 56, 57 den Leitungen "ZWEI" beziehungsweise "FÜNF" entsprechend aktiv geschaltet; die Summanden gelangen dann zum UND-Glied 51 und werden bei entsprechender Aktivierung der Multiplexer 58, 59 am Ausgang OUT abgegeben. Soll zum Beispiel eine NAND-Verknüpfung durchgeführt werden, so schaltet der Multiplexer 58 zum Inverter 54 und am Ausgang OUT steht dann das negierte UND-Ergebnis an.

Patentansprüche

1. Verfahren zum Betrieb einer Datenverarbeitungseinrichtung mit programmier- und konfigurierbarer Zellstruktur, wobei die Datenverarbeitungseinrichtung eine Zellmatrix aus einer Vielzahl orthogonal zueinander angeordneter, homogen strukturierter Zellen, welche in ihrer Funktion und Vernetzung durch eine Ladelogik frei programmierbar sind, enthält, **dadurch gekennzeichnet,** daß

a) ein Konfigurationsprogramm, bestehend aus einer Folge von Ladelogik-Befehlen, die jeweils die Funktion und Vernetzung der einzelnen Zellen angeben, so daß sich aus einer Teil folge eine spezielle Konfiguration mit spezieller(en) Anwendung(en) ergibt, vorhanden ist, auf das die Ladelogik (30) Zugriff hat (Fig. 3, Fig. 8, Fig. 14, Fig. 15) und das von ihr ausgeführt wird,

b) zunächst durch eine bestimmte Anzahl von Ladelogik-Befehlen eine Konfiguration (Fig. 20b) mit spezieller(en) Anwendung(en), mittels der Ladelogik (30) zu Beginn als Startkonfiguration in der Zell-Matrix (Fig. 2, Fig. 3) eingestellt wird,

c) durch eine Statemachine (45) oder einen Komparator (48) die Beendigung der Durchführung einer Anwendung einer oder mehrerer Zellen erkannt wird,

d) durch die Erkennung eine Rückmeldung an die Ladelogik (30) erfolgt, die die Programmabarbeitung des Konfigurationsprogramms mit einer anderen Teilfolge (Fig. 21b, Fig. 22b) fortsetzt, welche nur Zellen umkonfiguriert, also neu erstellt, die die Durchführung ihrer Funktion für die aktuelle Anwendung bereits beendet haben oder nicht benötigt werden, so daß die Abarbeitung der Datenströme der an der aktuellen Konfiguration noch beteiligten, parallel (z. B. Register 41, 42) oder gepipelined (z. B. Fig. 22a, Summierer 141 und Multiplizierer 149) arbeitenden Zellen, die sich noch in der Durchführung ihrer Anwendung befinden, nicht gestört wird.

2. Verfahren zum Betrieb einer Datenverarbeitungseinrichtung nach Anspruch 1, dadurch gekennzeichnet, daß durch eine Priorisierungs-Logik (z. B. Fig. 19) das Umkonfigurieren der unabhängigen parallel oder gepipelined arbeitenden Zellen, einer Funktion oder Konfiguration in der optimalen richtigen Reihenfolge über eine Rückmeldung an die Ladelogik sicherstellt wird.

3. Verfahren zum Betrieb einer Datenverarbeitungseinrichtung nach einem der Ansprüche 1 oder 2, dadurch gekennzeichnet, daß die Operanden aus mehreren, beliebig vielen unabhängig adressierten und gesteuerten Speichern (Fig. 20a, Fig. 20c; RAM1, RAM2) bezogen werden und das Ergebnis in einen dieser Speicher (Fig. 20a, Fig. 20c; RAM1) oder einen davon unabhängigen oder eine separaten Datenkanal (Fig. 14, Fig. 15; IO) ausgegeben wird.

4. Verfahren zum Betrieb einer Datenverarbeitungseinrichtung nach einem der Ansprüche 1 bis 3, dadurch gekennzeichnet, daß zur besseren Auslastung der Vernetzung von Zellen, Busse mit Hilfe von zwischengeschalteten Schaltern (Fig. 18; Treiber) Verwendung finden, so daß Busse in unabhängige Abschnitte aufgeteilt werden können.

Hierzu 13 Seite(n) Zeichnungen

– Leerseite –

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

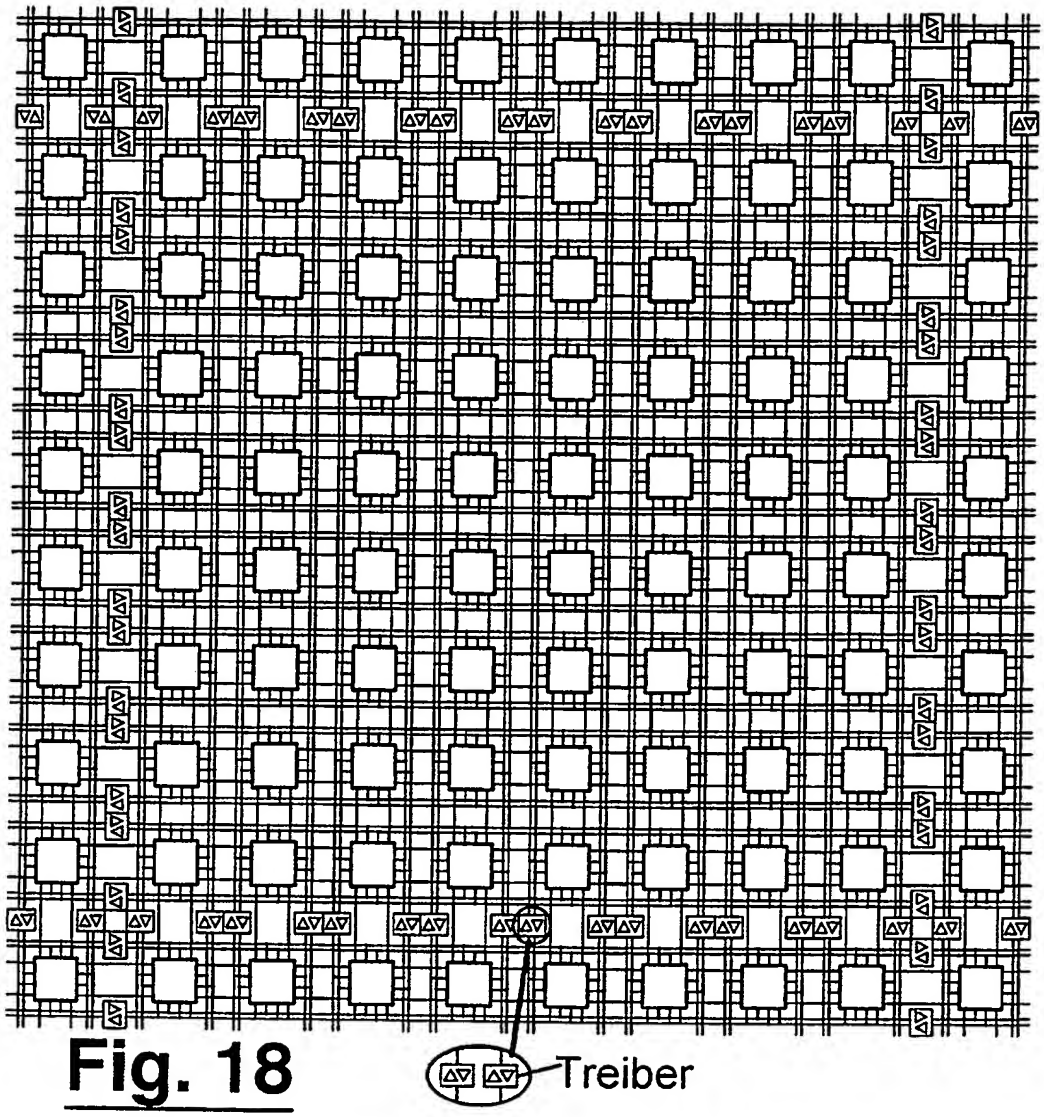


Fig. 18

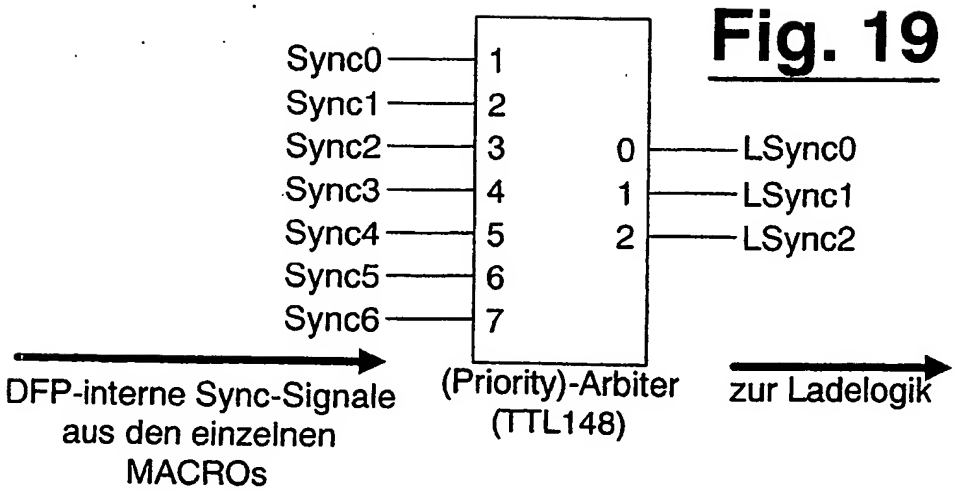


Fig. 19

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

Fig. 1

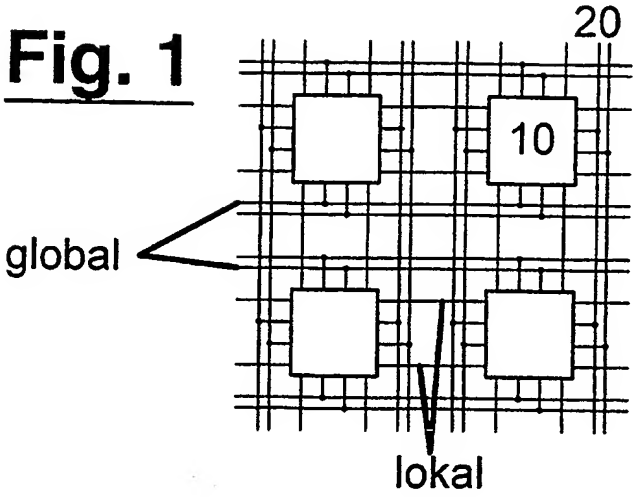


Fig. 2

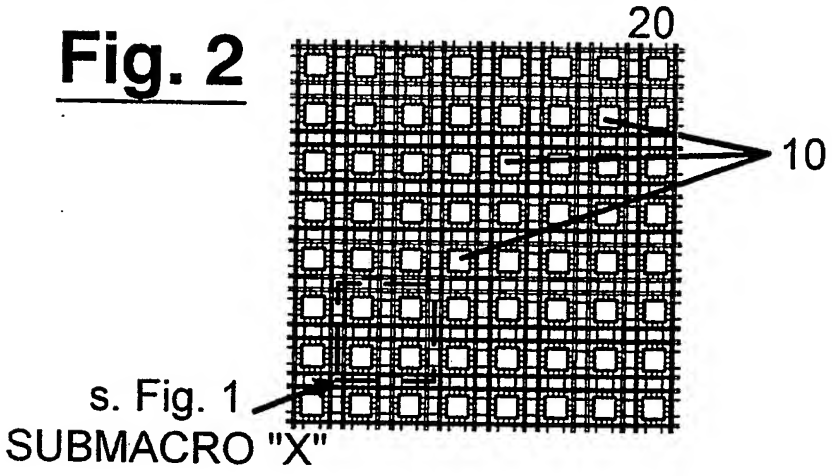
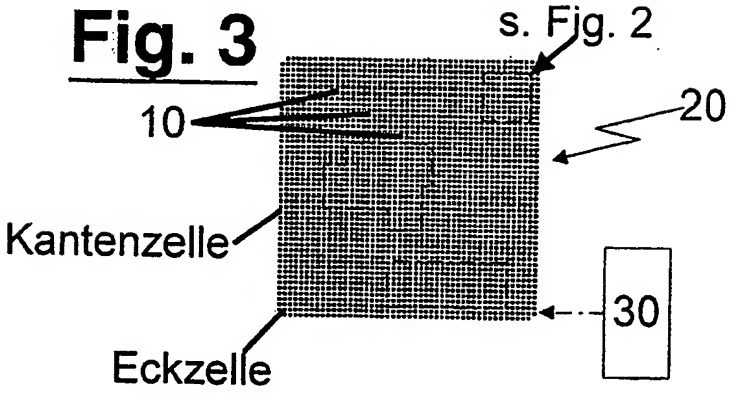


Fig. 3



Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

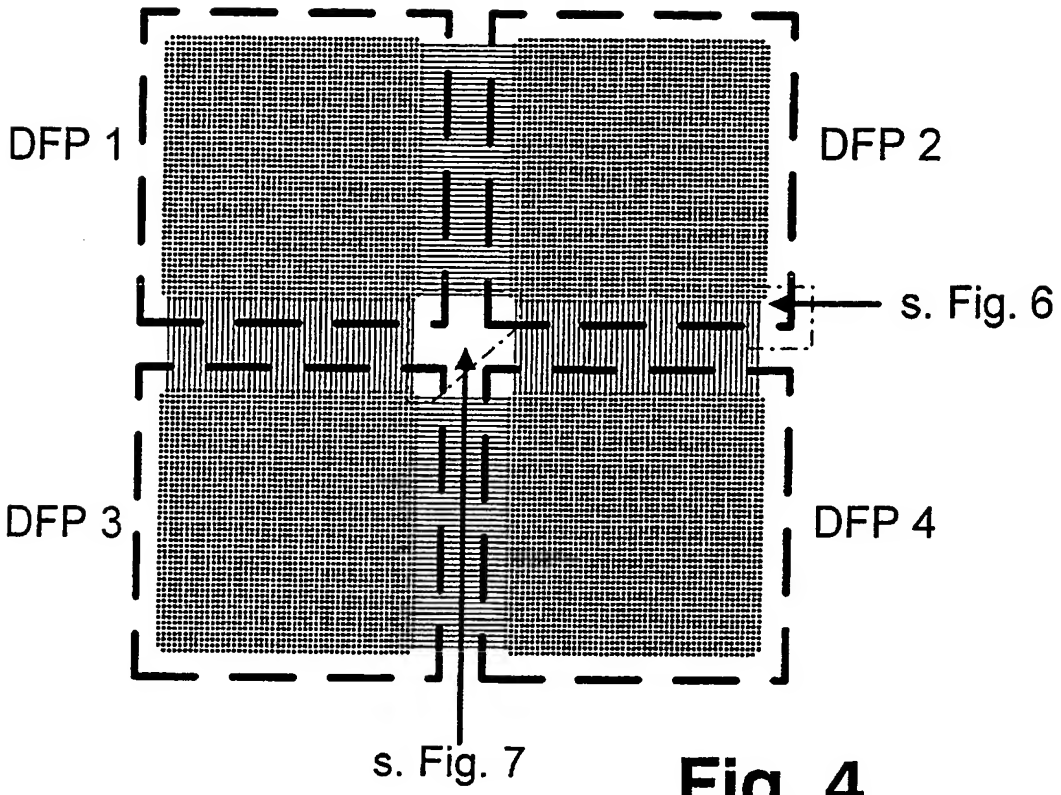


Fig. 4

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

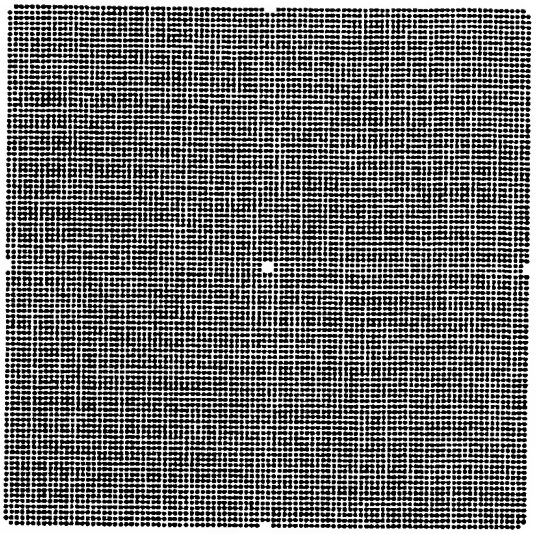


Fig. 5

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

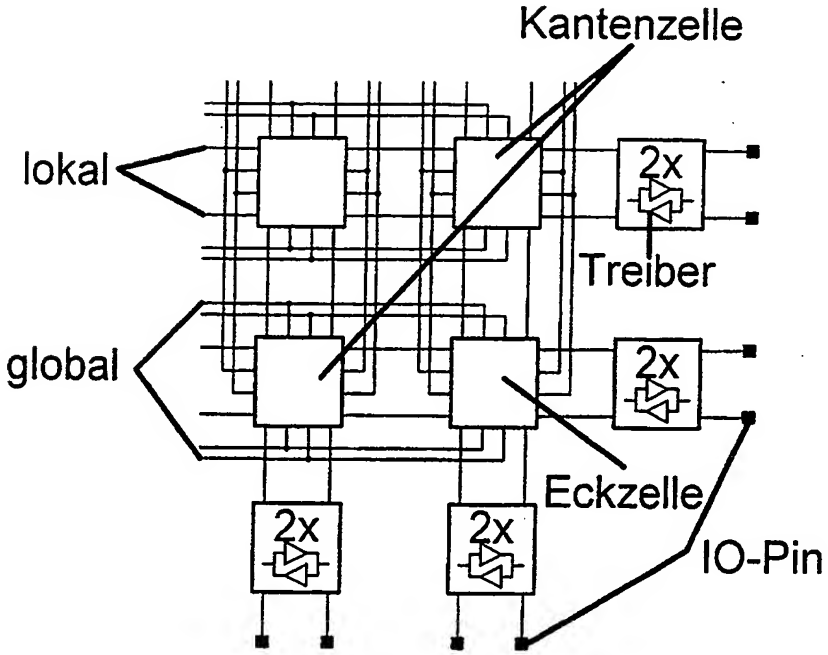


Fig. 6a

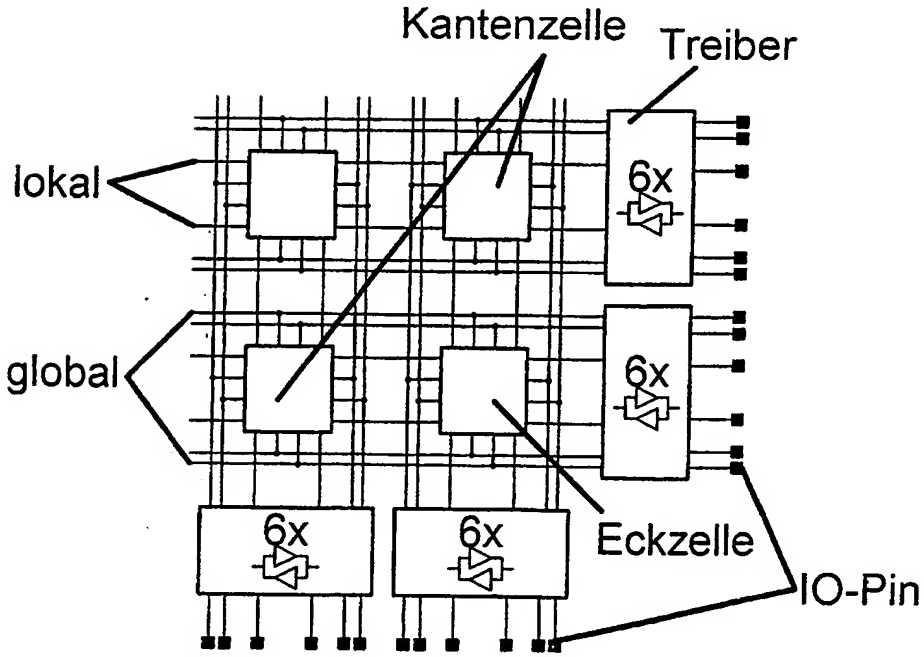


Fig. 6b

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

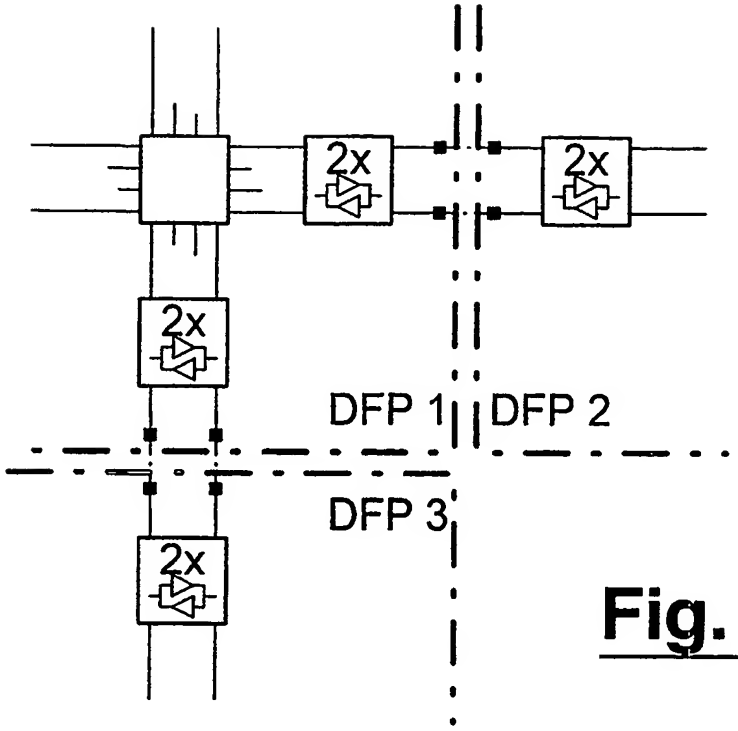


Fig. 7a

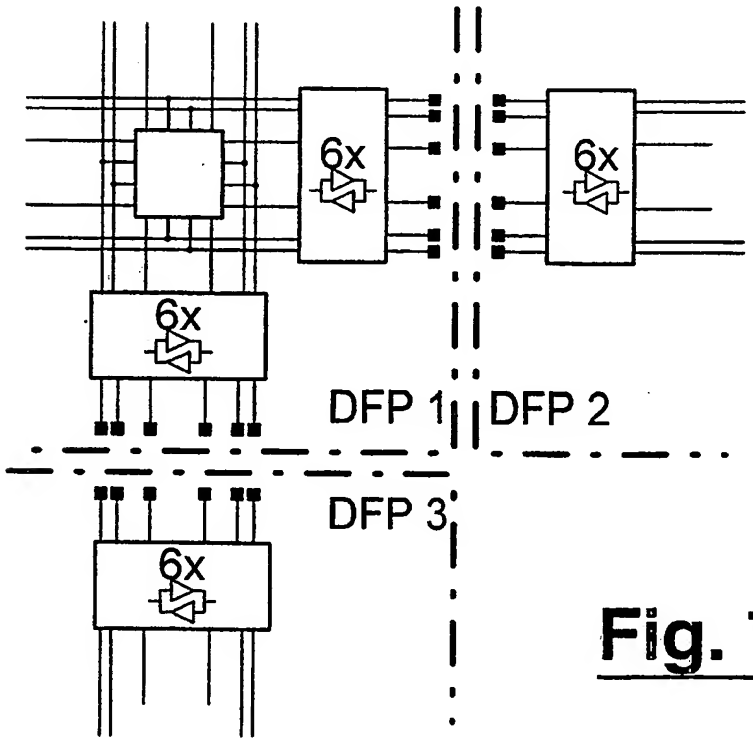
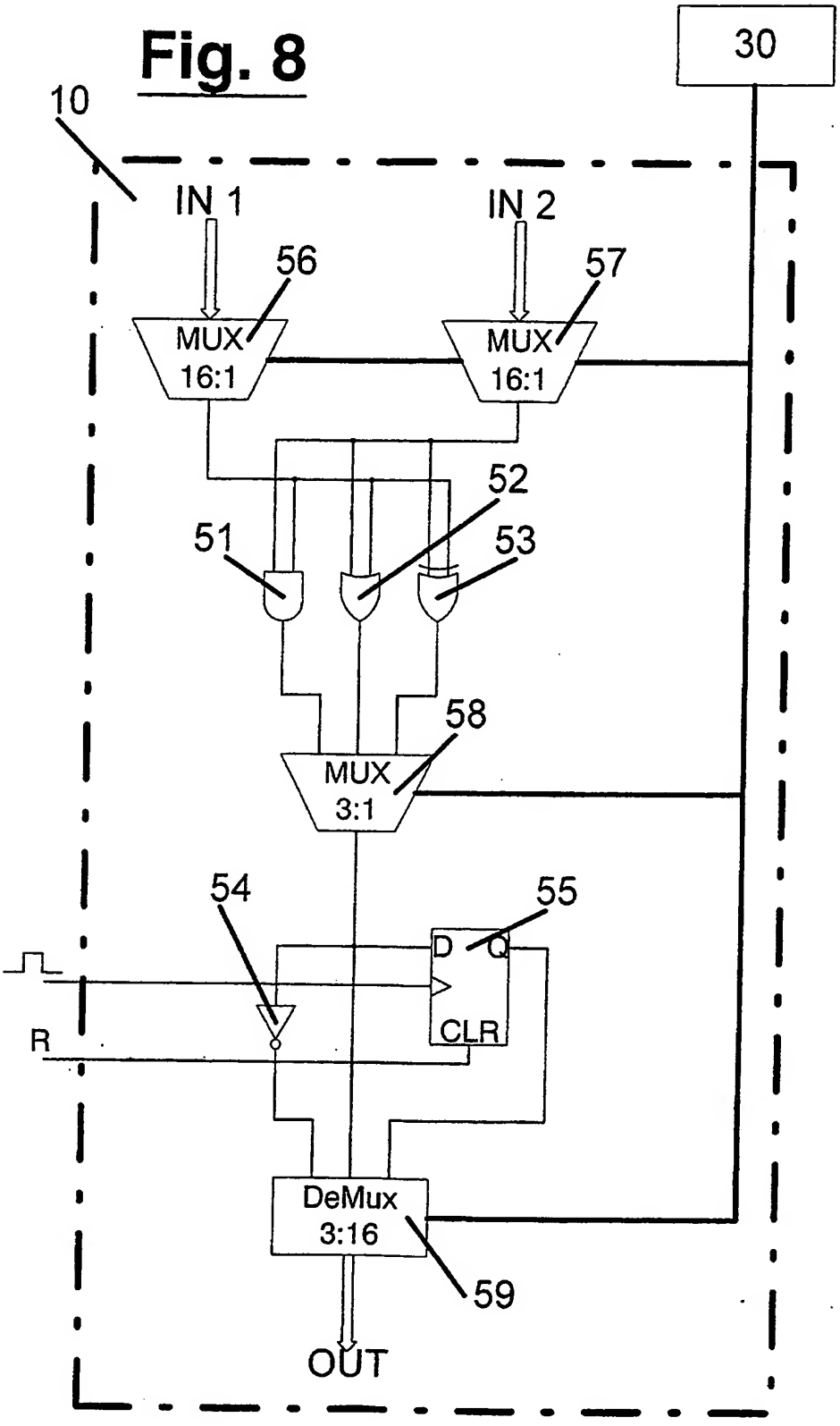


Fig. 7b

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

Fig. 8



Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

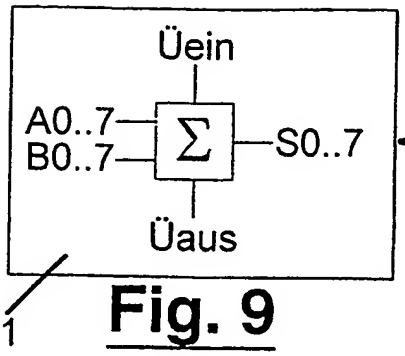


Fig. 9

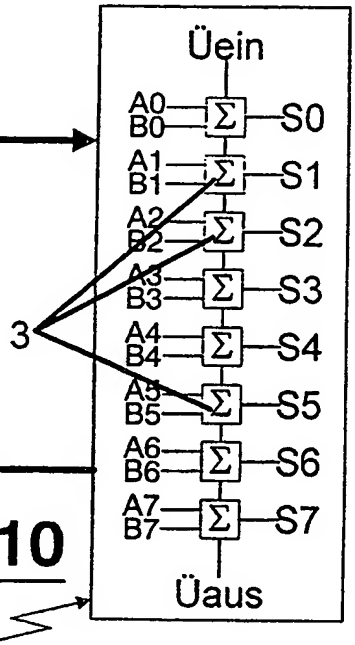


Fig. 10

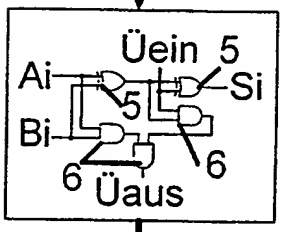


Fig. 11

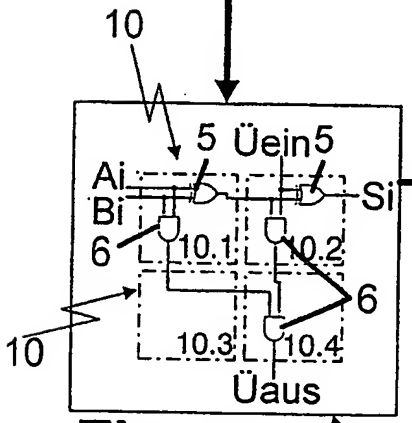


Fig. 12

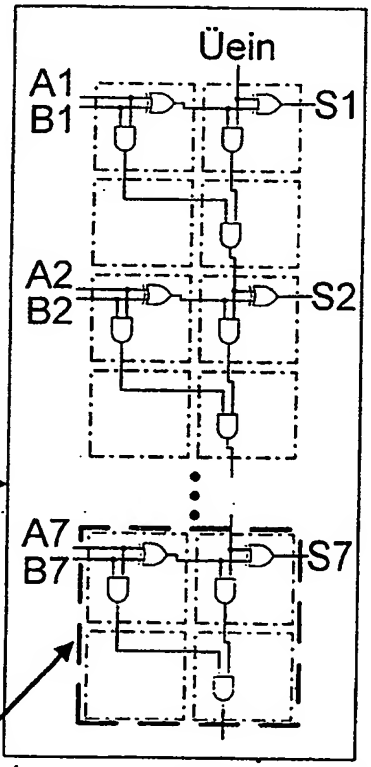


Fig. 13

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

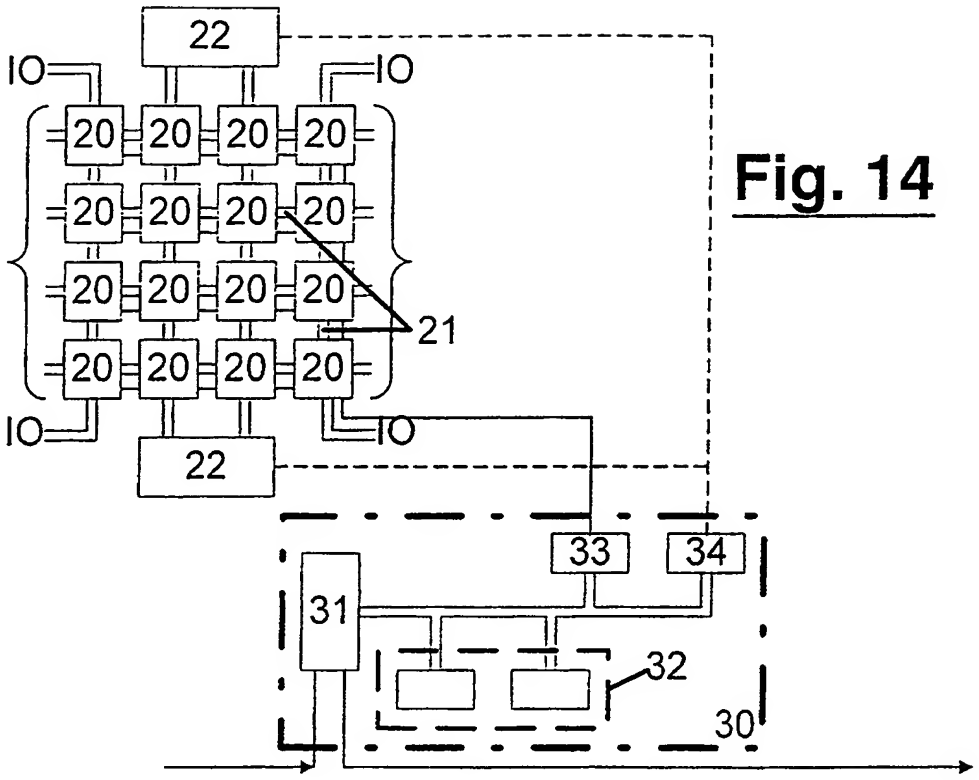


Fig. 14

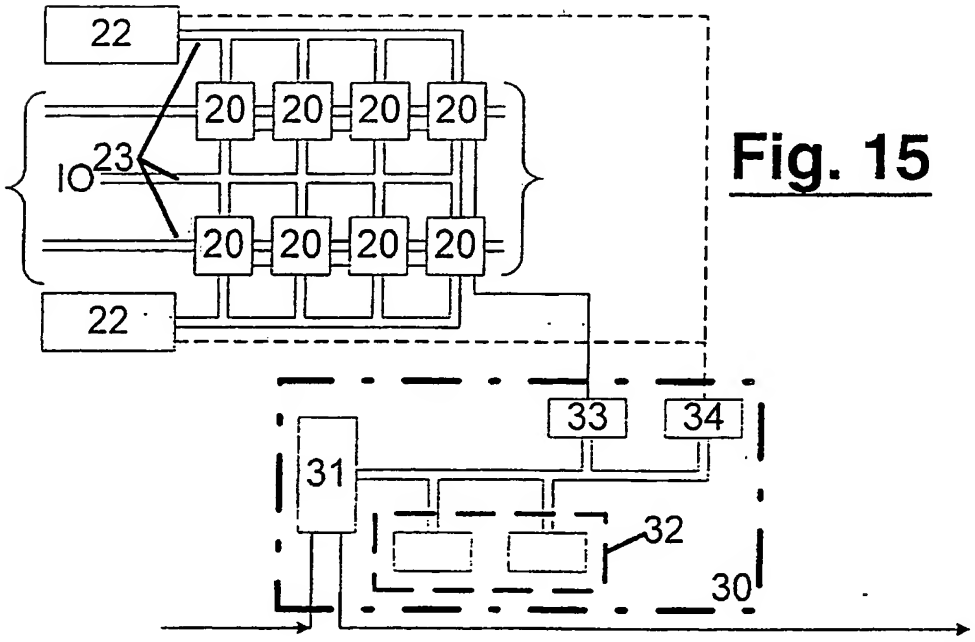


Fig. 15

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

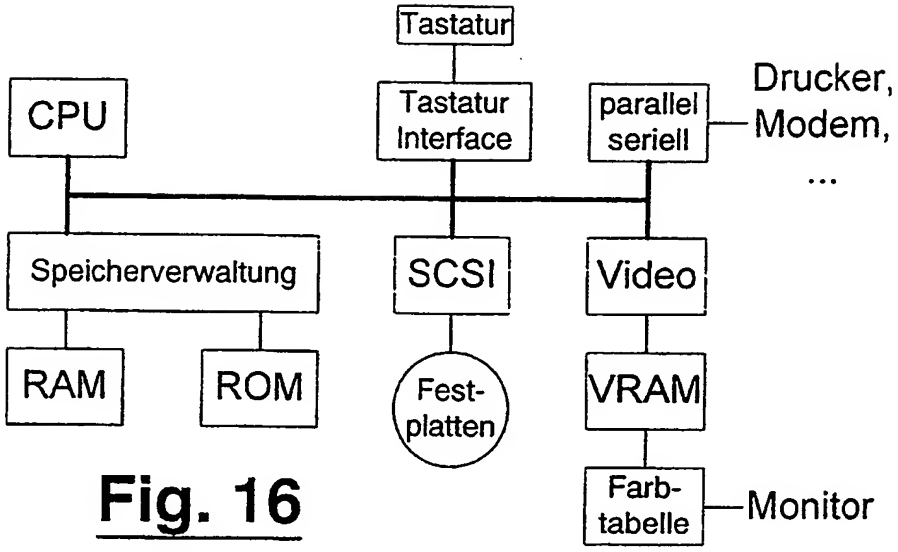


Fig. 16

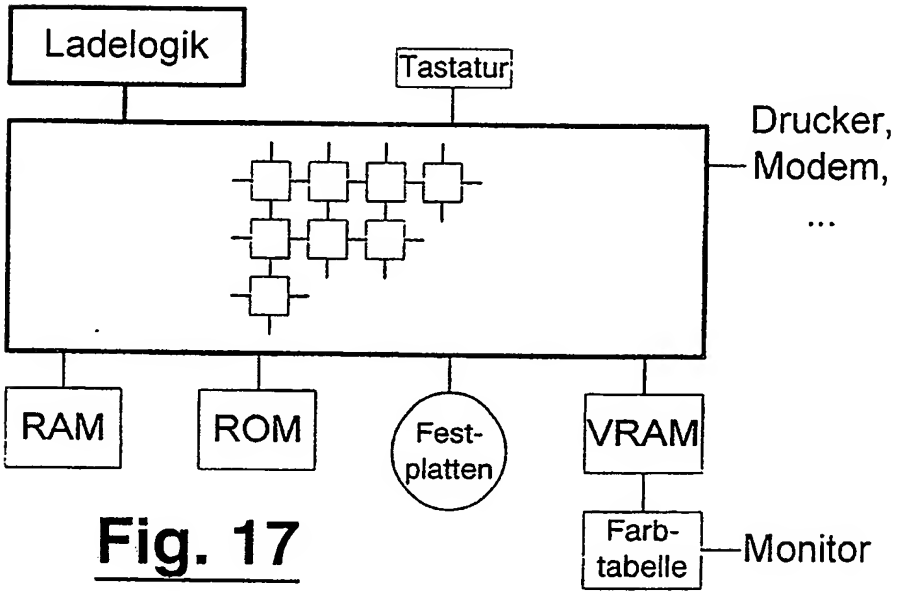


Fig. 17

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

Fig. 20a

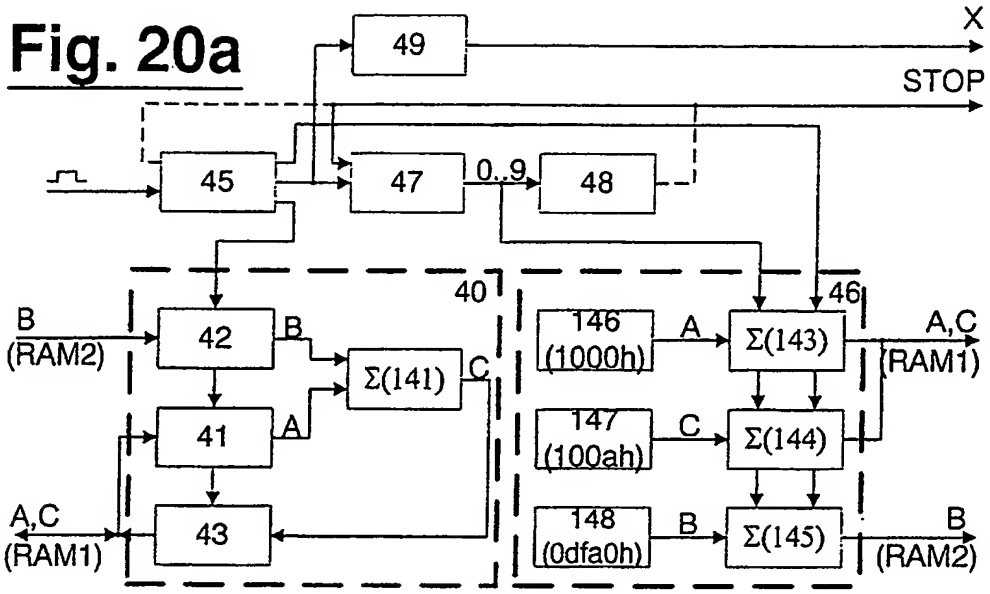
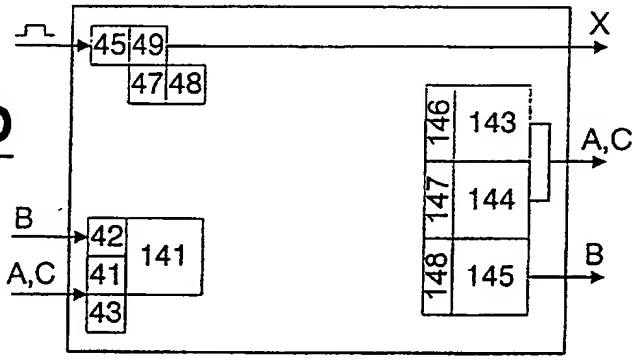
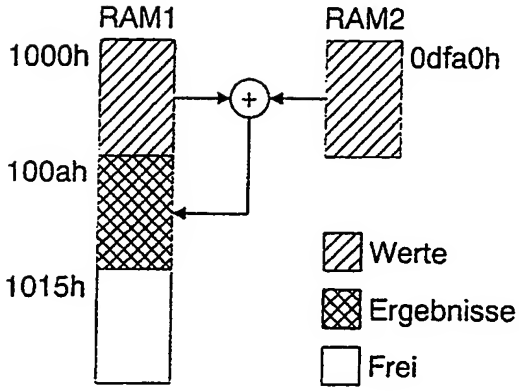


Fig. 20b





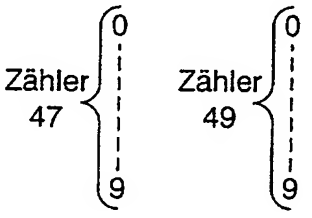


Fig. 20c

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

Fig. 21a

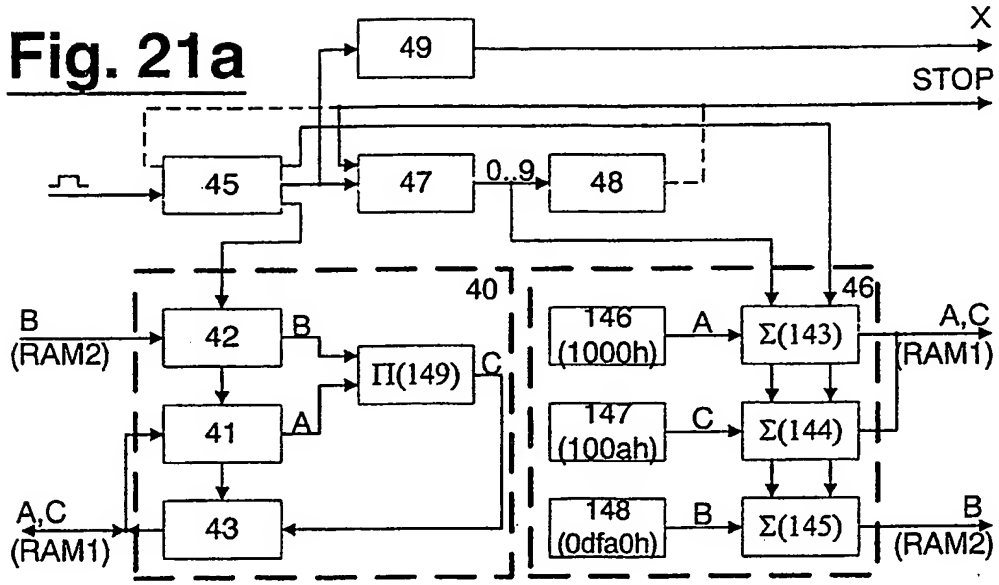
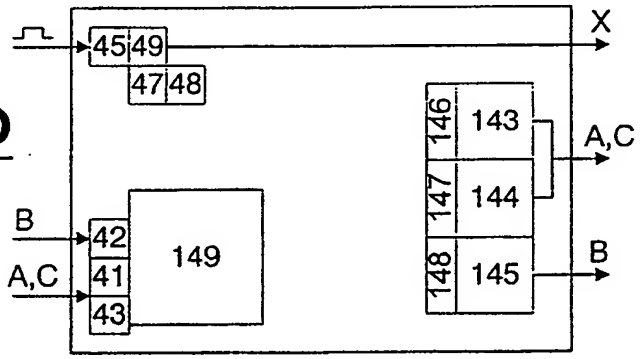
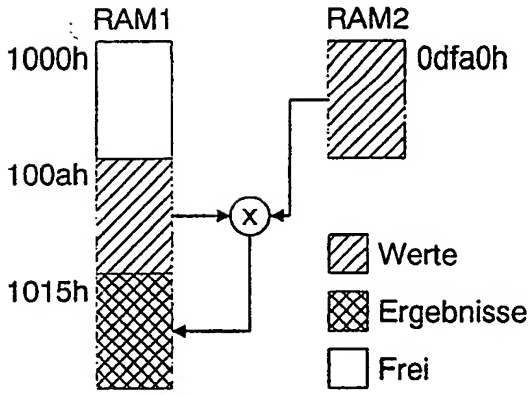


Fig. 21b





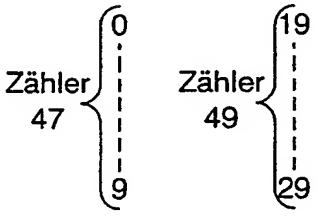


Fig. 21c

Nummer: DE 44 16 881 C2
Int. Cl.[6]: G 06 F 15/80
Veröffentlichungstag: 19. März 1998

Fig. 22a

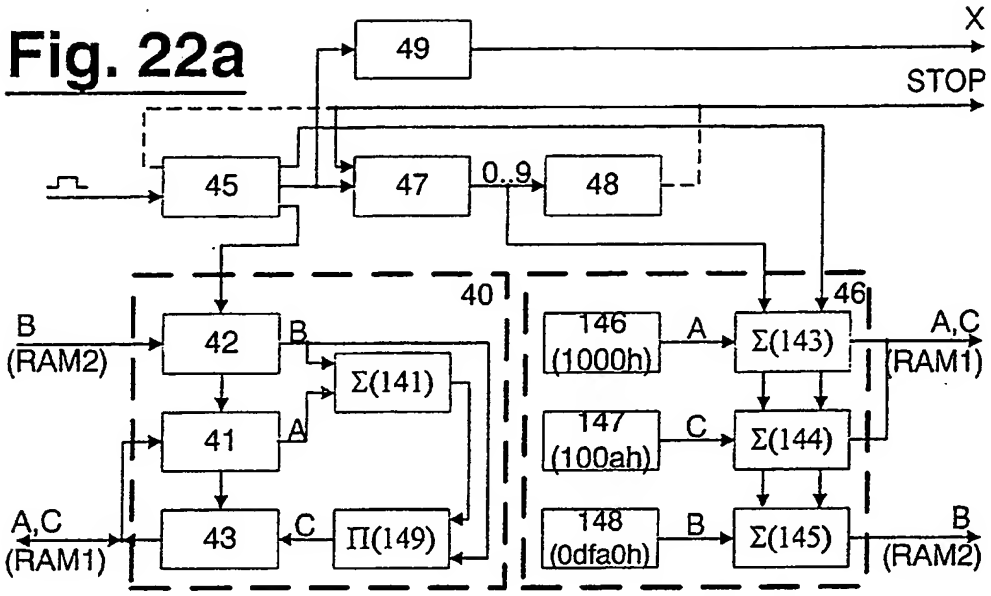
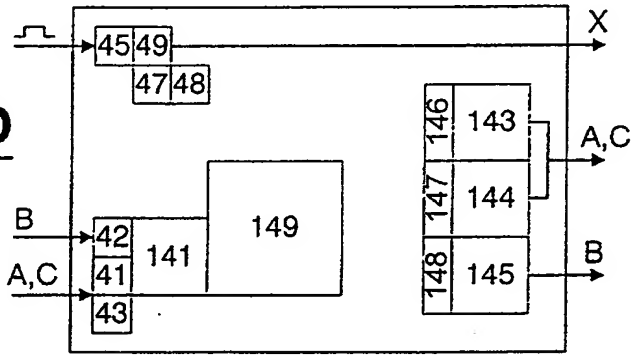
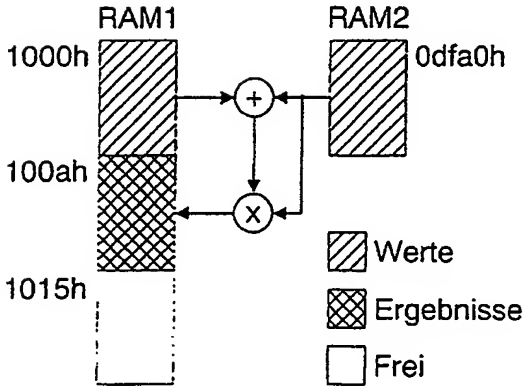


Fig. 22b





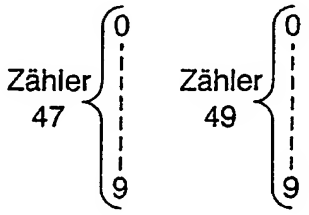


Fig. 22c